**Data Structures Practice Questions in JavaScript**

1. Write a function uniqueElements(arr) that takes an array and returns a new array containing only the unique elements from the original array, preserving their order.
2. Write a function firstNonRepeatingChar(str) that takes a string and returns the first character that does not repeat. If all characters repeat, return null.
3. Write a function isUnique(str) that takes a string and determines if it has all unique characters. Return true if it does, false otherwise. **What if you cannot use additional data structures?**
4. Write a function checkPermutation(a, b) that takes two strings, and determines if one is a permutation of the other (this means that they are composed of the same letters but in a different arrangement). Return true if they are, false otherwise.
5. Write a function URLify(str) that takes a string, and replaces all spaces in a string with ‘%40’, returning the newly modified string.
6. Write a function oneAway(str) that takes two strings, and determines if they are one edit (or zero edits) away (An edit is considered something like inserting a character, removing a character, or replacing one). Return true if they are one or zero edits away, false otherwise. **Hint, use a map.**
   1. pale, ple → true
   2. pale, bale → true
   3. pale, bake → false
7. Create a Stack class in JavaScript with the following methods:
   1. push(value): Adds a value to the top
   2. pop(): Removes and returns the top value
   3. peek(): Returns the top value without removing it
   4. isEmpty(): Returns true if the stack is empty
   5. size(): Returns the number of elements in the stack
   6. **Hint(s): use an array, and field to track the front of the stack**
8. Create a Queue class with the following methods:
   1. enqueue(value): Adds a value to the back
   2. dequeue(): Removes and returns the front value
   3. peek(): Returns the front value without removing it
   4. isEmpty(): Returns true if the queue is empty
   5. size(): Returns the number of elements in the queue
   6. **Hint(s): use an array, and field to track the front of the queue**
9. Create a LinkedList class in JavaScript with the following methods:
   1. append(value): Adds a value to the end
   2. prepend(value): Adds a value to the beginning
   3. delete(value): Removes the first occurrence of a value
   4. find(value): Returns true if the value exists in the list
   5. isEmpty(): Returns true if the list is empty
   6. size(): Returns the number of elements in the list
   7. **Hint(s):** 
      1. **use a Node class with value and next properties**
      2. **maintain a *head* field for tracking the start.**
      3. **Consider edge cases such as empty list, deleting the head, deleting a non-existent value.**
10. Create a BinarySearchTree class in JavaScript with the following methods:
    1. insert(value): Inserts a value in the correct position
    2. contains(value): Returns true if the value exists
    3. remove(value): Removes a node with a given value\*
    4. findMin(): Return the smallest value in the tree
    5. findMax(): Return the largest value in the tree
    6. **Hint(s):** 
       1. **use a Node class with value, left and right properties.**
       2. **Base cases for insert(), remove(), and contains() should handle the null case.**
       3. **Consider three cases for removal: Node has no children, Node has one child, Node has two children (replace with in-order successor)**
11. Create a HashTable class in JavaScript with the following methods:
    1. set(key, value): Stores a key-value pair
    2. get(key): Retrieves the value for a key
    3. remove(key): Deletes a key-value pair
    4. hasKey(key): Returns true if key exists
    5. size(): Returns the number of stored key-value pairs
    6. getKeys(): Returns the keys in the HashTable as an array
    7. getValues(): Returns the values in the HashTable as an array
    8. **Hint(s):** 
       1. **use an array of fixed size for storage (setting size in constructor to what the user wants)**
       2. **create a simple hash function that converts a key to an index (perhaps if the key is a string, multiplying the ASCII numeral for each letter together, then modulus by the size of the array)**
       3. **Handle collisions with linear probing (find next available slot), make sure to be able to wrap around to the beginning**
       4. **Consider resizing when the load factor exceeds a threshold (double size of hashTable, rehash everything), you can create a new method that does this, and is called when load factor exceeds a threshold!**